Le kd-Tree : une méthode de subdivision spatiale

Présentation de M2RI - Module CTR

Cédric Fleury

Université de Rennes 1 - INSA de Rennes

17 Janvier 2008

1 - Introduction

Le kd-Tree, abréviation pour « *k-dimensional tree* »

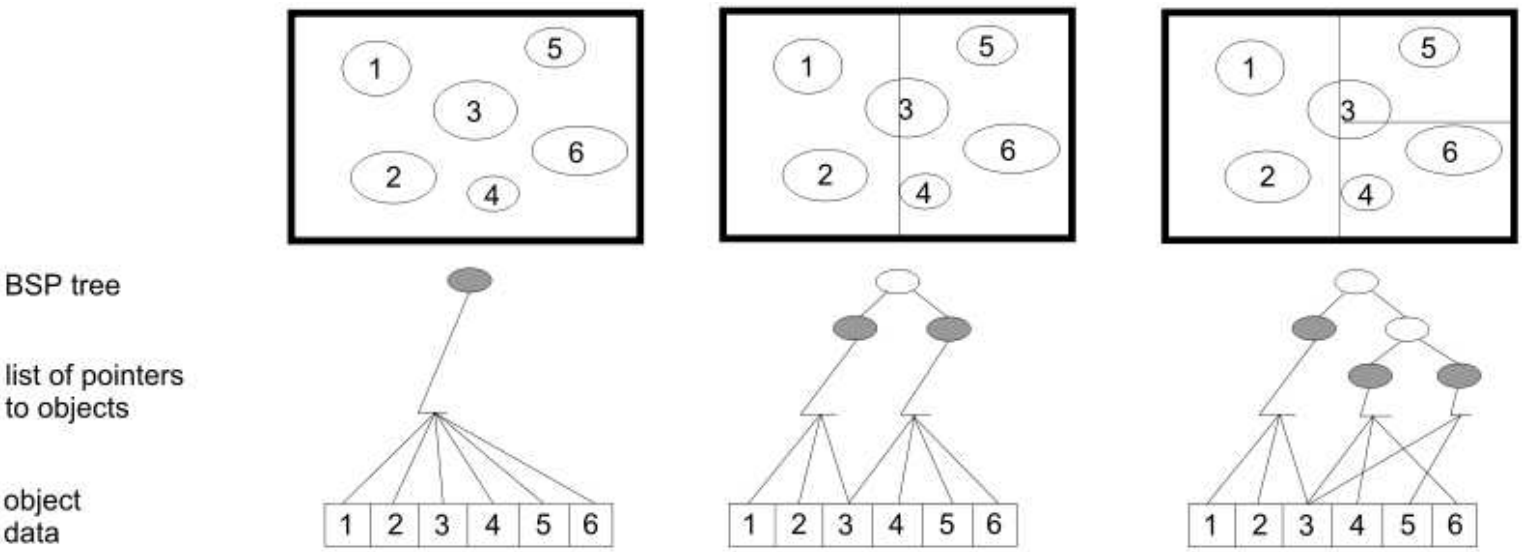
- Une structure pour organiser des données dans un espace à k-dimensions
- De nombreux types d'applications différents
  - accélérer la recherche de données dans un espace multi-dimensions
  - permettre la recherche d'intervalles ou de plus proches voisins
- Souvent utilisé dans les algorithmes de lancer de rayon

Plan

1. Introduction
2. Définition d'un kd-Tree
3. Construction d'un kd-Tree
   - Algorithme général
   - Méthodes basiques
   - Méthodes utilisant la « Surface Area Heuristic »
4. Intégration d'un kd-Tree dans un algorithme de lancer de rayon
   - Utilisation d'un kd-Tree pour le lancer de rayon
   - Algorithme de parcours d'un kd-Tree
5. Autres applications
6. Conclusion

2 - Définition d'un kd-Tree

- Le kd-Tree : un cas particulier des BSP trees (« *Binary Space Partitionning trees* »)
  - décompose l'espace en volumes englobants (ou voxels)
  - découpe chaque voxel en deux sous-voxels grâce à un plan séparateur
  - est représenté sous la forme d'un arbre binaire



- Particularité du kd-Tree : plans séparateurs toujours perpendiculaires aux axes du repère de l'espace

2 - Définition d'un kd-Tree

- Le kd-Tree a un rôle double :
  - organiser l'espace pour accélérer le traitement des données
  - structurer les données sous la forme d'un arbre binaire

- Le kd-Tree, dans le cas du lancer de rayon :
  - la racine $\Leftrightarrow$ la boîte englobante de toute la scène 3D
  - un noeud = un plan séparateur + deux fils correspondant aux deux sous-voxels
  - une feuille = la liste des objets contenus dans le voxel correspondant

Construction d'un kd-Tree

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - Si **non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - Si **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - **Si non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - **Si oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - **Si non** : Rendre la **Feuille(O)** contenant la liste des objets O
     - **Si oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - Si **non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - Si **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - Si **non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - Si **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - Si **non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - Si **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - Si **non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - Si **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - **Si non** : Rendre la **Feuille**(O) contenant la liste des objets O
     - **Si oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.1 - Construction : algorithme général

- Un algorithme récursif :

  recConst(voxel V, objets O)

  1. Tester s'il faut diviser le voxel courant V :
     - **Si** **non** : Rendre la **Feuille(O)** contenant la liste des objets O
     - **Si** **oui** : Continuer
  2. Trouver le bon plan séparateur p
  3. Couper le voxel V avec p pour obtenir V_G et V_D
  4. Repartir les objets O dans V_G et $V_D \Rightarrow O_G$ et O_D
  5. Rendre le **Noeud**(p, *recConst*(V_G,O_G), *recConst*(V_D,O_D))

- L'aglorithme est initialisé avec toute la scène

- Les méthodes de construction diffèrent par :
  - la façon de choisir le plan séparateur (étape n° 2)
  - la façon de terminer l'algorithme (étape n° 1)

3.2 - Construction : méthodes basiques

- Le choix du plan séparateur
  - **son orientation** : perpendiculaire à chacun des axes du repère à tour de rôle
  - **sa position** : 2 méthodes possibles
    - couper le voxel au milieu par le plan médian
    - couper le voxel au niveau de l'objet « médian »
- La fin de l'algorithme
  - nombre de objets dans le voxel inférieur à un seuil $\mathcal{S}_{minObj}$
  - nombre de subdivisions supérieur à un seuil $\mathcal{S}_{maxProf}$

3.3 - Construction : Méthodes utilisant la SAH

La SAH : « Surface Area Heuristic »

- permet d'estimer le coût de parcours du kd-Tree lors de sa construction à partir de $\mathcal{K}_t$ et de $\mathcal{K}_i$
- définit :
  - la probabilité conditionnelle qu'un rayon intersecte un sous-voxel $V_{sous} \in V$ sachant qu'il intersecte le voxel V :

$$\mathcal{P}_{[V_{sous}|V]} = \frac{Volume(V_{sous})}{Volume(V)}$$

  - le coût de parcours d'un voxel V coupé par un plan p :

$$\mathcal{C}_V(p) = \mathcal{K}_t + \mathcal{P}_{[V_g|V]}\mathcal{C}_{V_g} + \mathcal{P}_{[V_d|V]}\mathcal{C}_{V_d}$$

  - le coût de parcours d'une feuille contenant $|O|$ objets :

$$\mathcal{C}_{feuille} = |O|\mathcal{K}_i$$

3.3 - Construction : Méthodes utilisant la SAH

- Wald et Havran [Havran06] proposent d'utiliser une approximation locale pour estimer le coût :

$$\mathcal{C}_V(p) \approx \mathcal{K}_t + \mathcal{P}_{[V_g|V]}|O_g|\mathcal{K}_i + \mathcal{P}_{[V_d|V]}|O_d|\mathcal{K}_i$$

- Le choix du plan séparateur :
  - trouver le plan qui minimise la fonction de coût $\mathcal{C}_V(p)$

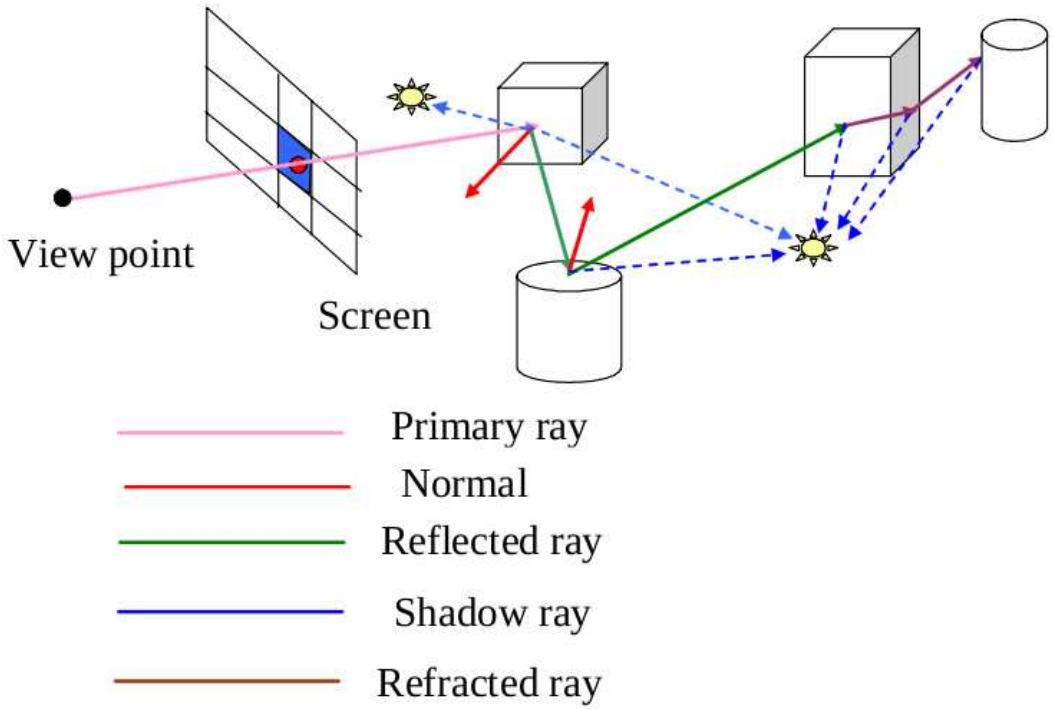
- La fin de l'algorithme :

$$terminer(V, O) = min_p\, \mathcal{C}_V(p) > |O|\mathcal{K}_i$$

Intégration d'un kd-Tree dans un algorithme de lancer de rayon

4.1 - Intégration : utilisation pour le lancer de rayon

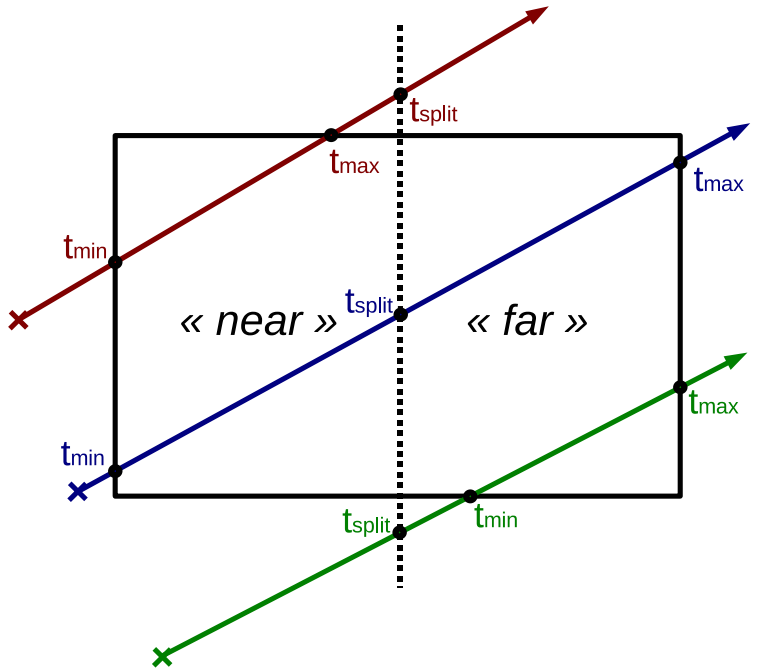
- Petit rappel sur le lancer de rayon (extrait de [Cours K.Bouatouch])



- L'utilisation d'un kd-Tree pour le lancer de rayon
  - structure la scène 3D à synthétiser
  - accélére le calcul des intersections entre les rayons et la scène

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

- Une des méthodes de parcours présentait par Havran [Havran00]
- Un algorithme récursif qui :
  - parcourt les nœuds et les feuilles du kd-Tree dont les voxels sont traversés par le rayon
  - détermine pour chaque nœud :
    - le fils le plus proche (« near ») et le plus éloigné (« far ») de l'origine du rayon
    - si les deux fils sont traversés par le rayon ou seulement un seul



4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - Cas **le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - Cas **le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - Cas **les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     - Tester si une intersection a été trouvé dans le fils « near »
       - oui Rendre cette intersection
       - non reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - Cas **le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - Cas **le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - Cas **les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     - Tester si une intersection a été trouvé dans le fils « near »
       - oui Rendre cette intersection
       - non reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     - Tester si une intersection a été trouvé dans le fils « near »
       - oui Rendre cette intersection
       - non reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - Cas **le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - Cas **le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - Cas **les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     - Tester si une intersection a été trouvé dans le fils « near »
       - oui Rendre cette intersection
       - non reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - Cas **le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - Cas **le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - Cas **les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     - Tester si une intersection a été trouvé dans le fils « near »
       - oui Rendre cette intersection
       - non reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

4.2 - Intégration : un algorithme de parcours d'un kd-Tree

Algorithme récursif de parcours :

recParcours(Voxel V, rayon r)

1. Tester si V est une feuille :
   - **Si oui** : Rendre la plus proche intersection entre r et les objets de V
   - **Si non** : Continuer
2. Trouver le fils « near » et le fils « far »
3. Tester quels sont les fils traversés par la rayon
   - **Cas le fils « near »** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
   - **Cas le fils « far »** $\Rightarrow$ reitérer : recParcours(V_{far}, r)
   - **Cas les deux fils** $\Rightarrow$ reitérer : recParcours(V_{near}, r)
     Tester si une intersection a été trouvé dans le fils « near »
     - **Si oui** Rendre cette intersection
     - **Si non** reitérer : recParcours(V_{far}, r)

5 - Autres applications du kd-Tree

- La détection de collisions [Kozak07]
  - accélérer le calcul des collisions des objets en mouvement avec la scène
  - efficace que dans certains cas particuliers

- Comparer des données
  - comparer les décompositions en kd-Tree
  - avoir une approche plus structurelle des données

  Ex : comparaison d'images d'astéroïdes [Kubica05]

- Organiser des données multi-dimensions
  - classer les données selon leur vecteur de caractèristiques à n-dimensions
  - faciliter la recherche de données ayant des caractéristiques proches

  Ex : extraction d'information dans des morceaux de musique [Reiss01]

6 - Conclusion

- Le kd-Tree
  - une structuration spatiale de l'espace à k-dimensions
  - un arbre binaire qui structure les données
- Une solution de plus en plus choisie pour le lancer de rayon
  - pour accélérer le calcul des intersections de manière importante
  - grâce à une complexité de construction et de parcours relativement faible
- Dans un cadre plus générale, le kd-Tree
  - permet d'accélérer le traitement de données multi-dimensions
  - a l'avantage d'être plus ou moins précise selon la concentration de données dans les différentes zones de l'espace

Références


[Havran06] I. Wald and V. Havran.
On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N).
Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, p61-69, 2006.


[Havran00] V. Havran.
Heuristic Ray Shooting Algorithms.
Faculty of Electrical Engineering, Czech Technical University, Prague, Novembre 2000.


[Cours K.Bouatouch] K. Bouatouch.
Cours : Ray Tracing
`http://www.irisa.fr/prive/kadi/Cours_LR2V/RayTracing.pdf`


[Kozak07] M. Kozak.
Collision.
`http://www.screamyguy.net/collision/`, Octobre 2007.


[Kubica05] J. Kubica, A. Moore, A. Connolly and R. Jedicke.
A Multiple Tree Algorithm for the Efficient Association of Asteroid Observations.
The 9th ACM Int. Conf. on Knowledge Discovery and Data Mining, p138-146, Août 2005.


[Reiss01] J. Reiss, J.-J. Aucouturier and M. Sandler.
Efficient multidimensional searching routines for music information retrieval.
The 2nd Annual Int. Symposium on Music Information Retrieval, p163-171, 2001.

Annexe - Plans « candidats » pour la construction par SAH